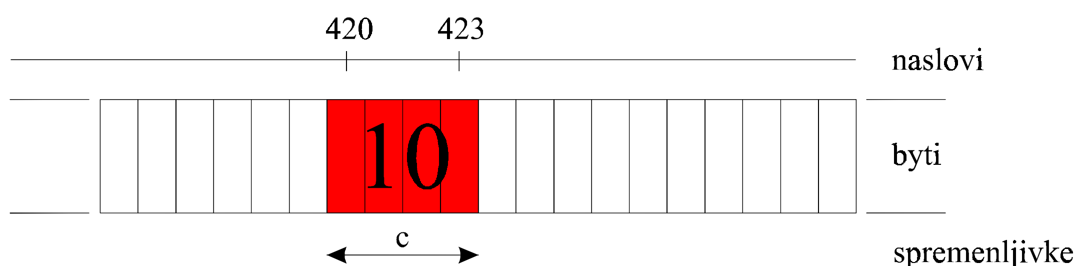
Izpeljani tipi

Izpeljani tipi so:

- *kazalci* na objekte nekega tipa,
- *polja* objektov nekega tipa,
- *strukture*, ki vsebujejo spisek objektov,
- *unije*, ki vsebujejo enega od objektov v spisku,
- *funkcije*, ki vrnejo objekt določenega tipa.

Kazalci

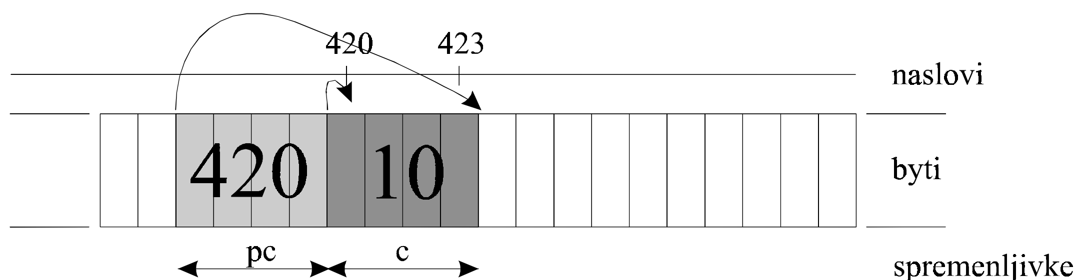
Kazalci so spremenljivke, ki vsebujejo naslov neke druge spremenljivke ali dela pomnilnika. Podobno kot v zvezi z običajnimi spremenljivkami tudi o kazalcu vemo ime, tip (ki pove, na kakšne vrste podatkov kaže) in način hranjenja.



Slika 1

Slika 1:
Spremenljivka c je na naslovih od 420 do 423. Če v programu napišemo npr. c = 10, potem s tem mislimo, naj se ta vrednost zakodira v del pomnilnika, ki je označen s c, torej v byte med naslovi 420 in 423.

Vzemimo, da imamo še neko drugo spremenljivko npr. z imenom pc. Želimo, da bi kazala na del pomnilnika, kjer je zapisan c. Torej mora biti v njej zakodiran podatek »420 do 423«, ali pa samo 420, prevajalnik pa mora vedeti, da je na naslovu 420 nekaj tako velikega, da se razteza do vključno 423. V pomnilniku dobimo npr. tako sliko:



Slika 2

Slika 2:
Del pomnilnika s kazalcem pc, ki kaže na c. Slučajno sta na zaporednih lokacijah v pomnilniku.

Da bi to lahko ukazali v programu, moramo znati:
- definirati kazalec,
- postaviti kazalec na neko vrednost,
- delati s kazalcem,
- delati z vrednostjo, na katero kazalec kaže.

V nadaljevanju si bomo ogledali vse osnove in postopke, ki jih moramo znati. Sedaj pa si oglejmo predstavitev kazalca še na drug način:

Kaj se zgodi pri definiciji spremenljivke?
Pri definiciji spremenljivke navedemo ime in tip spremenljivke in prevajalnik za spremenljivko rezervira pomnilniški prostor na določenem naslovu. Vrednost spremenljivke je vrednost, ki je shranjena na tem naslovu, tip pa določa zalogo vrednosti spremenljivke. Tako je vsaka spremenljivka v nekem trenutku določena z imenom, tipom, naslovom in vrednostjo. Če definiramo in inicializiramo spremenljivko, na primer `int stevilo = 12`, lahko to pomnilniško lokacijo predstavimo kot škatlico, pred katero zapišemo ime, nad njo naslov, pod njo tip, v škatlico pa vrednost (slika 3).

```
         0x0012FF80
stevilo [    12    ]
         int
```

Slika 3 (slika spremenljivke)

Vrednost spremenljivke dosegamo z njenim imenom, naslov spremenljivke pa dosegamo z operatorjem &. Operator & (naslovni operator) pred imenom spremenljivke vrne njen naslov. Oglejmo si program, v katerem izpišemo spremenljivko in njen naslov. Naslovi se izpišejo v šestnajstiškem številskem sistemu.

```
// Program 1
#include <stdio.h>

int main()
{
   int stevilo = 12; // celostevilcna spremenljivka

   printf("Vrednost spremenljivke: %d\n",  stevilo);
   printf("Naslov spremenljivke  : 0x%X", &stevilo);

   return 0;
}
```

Program izpiše naslednje:

```
Vrednost spremenljivke: 12
Naslov spremenljivke  : 0x63FE00
```

Kazalec (pointer) je taka spremenljivka, katere vrednost je naslov pomnilniške lokacije. To omogoča, da obravnavamo naslov pomnilniške lokacije kot spremenljivko.

Kako definiramo kazalec?
Oglejmo si, kako definiramo kazalec oz. spremenljivko kazalčnega tipa. Za lažje razpoznavanje kazalcev v programu pišemo identifikatorje kazalcev s črko k in podčrtajem (k_).

Če imamo spremenljivko tipa `int`, potem določimo kazalec nanjo tipa `int*`. Oglejmo si program, kjer definiramo nekaj kazalcev.

```
// Program 2
#include <stdio.h>

int main()
{
   int stevilo = 12; // celostevilcna spremenljivka
   int *k_stevilo;   // kazalec na int

   long *k_kaz1;     // kazalec na long
   double *k_kaz2;   // kazalec na double

   printf("Velikost tipa int*        : %d\n", sizeof(int*));
   printf("Velikost kazalca k_stevilo: %d\n", sizeof(k_stevilo));
   printf("Velikost kazalca k_kaz1   : %d\n", sizeof(k_kaz1));
   printf("Velikost kazalca k_kaz2   : %d",   sizeof(k_kaz2));

   return 0;
}
```

```
Velikost tipa int*         : 4
Velikost kazalca k_stevilo : 4
Velikost kazalca k_kaz1    : 4
Velikost kazalca k_kaz2    : 4
```

Z zapisom `int *k_stevilo` definiramo kazalec `k_stevilo`. Znak * na tem mestu označuje, da je definirana spremenljivka kazalčnega tipa. Operatorju * pravimo dereferenčni operator, operaciji, ko izračunamo, na kaj kazalec kaže, pa dereferenciranje kazalca. Ustvarimo lahko kazalce, ki kažejo na poljuben tip. Pri definiciji kazalca se, kot pri vsaki spremenljivki, rezervira pomnilniški prostor in kazalci ponavadi zasedejo 4 zloge. Ker vsebuje vsak kazalec naslov, je velikost kazalca vedno enaka in izraza `sizeof(int*)` in `sizeof(double*)` vrneta enako vrednost. Kazalec, ki ga ne inicializiramo, vsebuje naključno vrednost!

Kako priredimo vrednost kazalcu?
Kazalcu priredimo vrednost tako, da vanj zapišemo naslov spremenljivke. Zato uporabimo operator &. Kazalcu lahko priredimo drug kazalec istega tipa. Če je kazalec tipa void, mu lahko priredimo katerikoli drug tip kazalca in ga lahko priredimo kateremu koli drugemu tipu. Kazalci tipa `void` kažejo na »prazni« del pomnilnika, kjer so samo surovi byti. Deklariramo ga tako:

```
void *p;
```

V zvezi s kazalci na podatke je dovoljeno samo prištevanje (+) in odštevanje (-) števil ter povečevanje (++) ali zmanjševanje (--) kazalca za eno. V zvezi s kazalci na funkcije ni dovoljeno računanje. Kazalci se pri odštevanju in seštevanju ne obnašajo kot števila, ampak povečevanje kazalca za 1 pomeni, da kazalec poslej kaže tik za objekt, na katerega je kazal prej.

Poglejmo si primer prireditve vrednosti kazalcu:

```
// Program 3
#include <stdio.h>

int main()
{
   int stevilo = 12;       // celostevilcna spremenljivka
   int *k_stevilo;         // kazalec na int
   k_stevilo = &stevilo;   // kazalec dobi vrednost

   printf("Vrednost spremenljivke stevilo  : %d\n",      stevilo);
   printf("Naslov spremenljivke   stevilo  : 0x%X\n\n", &stevilo);

   printf("Vrednost spremenljivke k_stevilo: 0x%X\n",  k_stevilo);
   printf("Naslov spremenljivke   k_stevilo: 0x%X",   &k_stevilo);

   return 0;
}
```

```
Vrednost spremenljivke stevilo  : 12
Naslov spremenljivke   stevilo  : 0x63FE00

Vrednost spremenljivke k_stevilo: 0x63FE00
Naslov spremenljivke   k_stevilo: 0x63FDFC
```

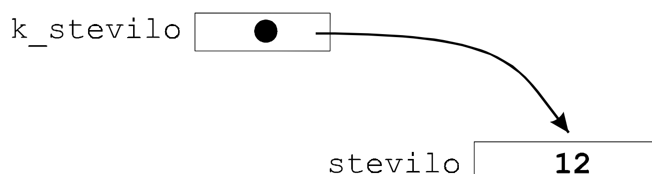
Kazalcu `k_stevilo` smo priredili naslov spremenljivke `stevilo` in pravimo, da kazalec »kaže na spremenljivko `stevilo`« ali da »naslavlja spremenljivko `stevilo`«. Predstavitev kazalca si še enkrat oglejmo na sliki 4. Na sliki sta prikazani spremenljivki `stevilo` in `k_stevilo`, ki sta rezervirani v pomnilniku druga za drugo.

```
            0x0063FE00
stevilo    [    12    ]
            int

            0x0063FDFC
k_stevilo  [0x0063FE00]
            int*
```

Slika 4 (kazalec ima vrednost)

Spremenljivka `stevilo`, ki je na naslovu `0x0063FE00`, je tipa `int` in je velika 4 zloge. Sledi kazalec `k_stevilo`, ki je na naslovu `0x0063FDFC`, za štiri zloge odmaknjen od spremenljivke `stevilo`. Kadar spreminjamo kazalec `k_stevilo`, spreminjamo vrednost na naslovu `0x0063FDFC` in tako dosežemo, da kaže na različne spremenljivke.

Vrednost kazalca je naslov pomnilniške lokacije, ki nas običajno ne zanima, zato je za programerja bolj pregledna naslednja slika:



Slika 5 (kazalec »kaže« na spremenljivko)

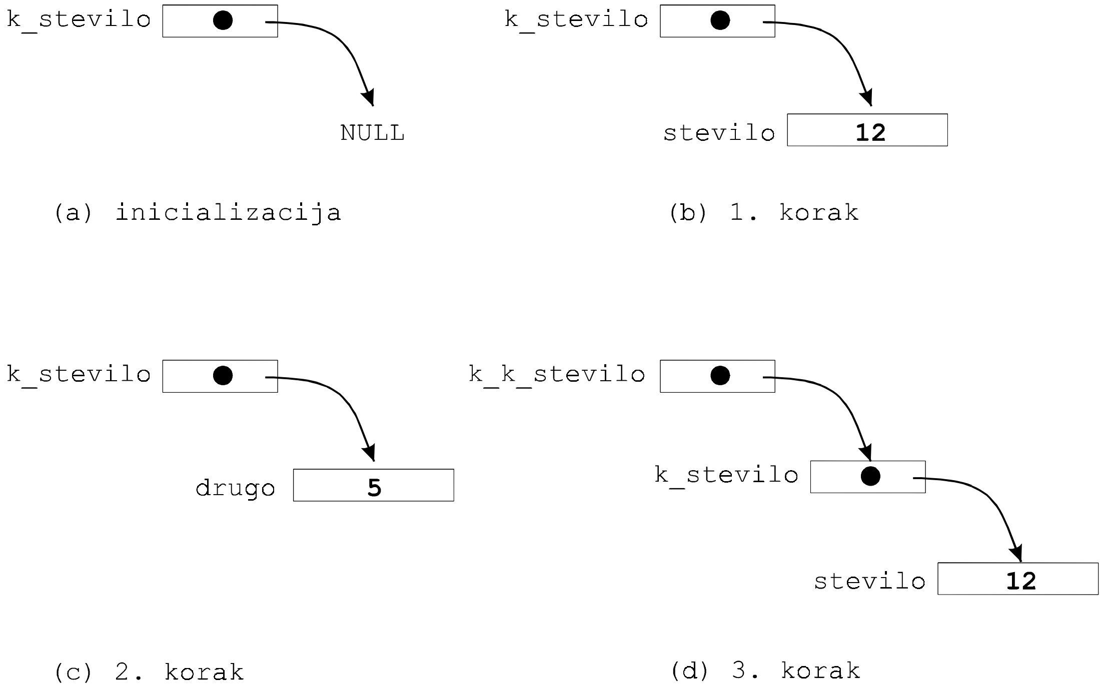
Kazalec ne smemo uporabljati, če ni inicializiran. Uporaba neinicializiranega kazalca lahko povzroči napako pri izvajanju programa. Za inicializacijo kazalcev uporabimo konstanto 0, simbol NULL ali pa naslov spremenljivke. Simbol NULL je podan v različnih vključitvenih datotekah, kot so `stdio.h`, `iostream.h`, ... Kazalec, ki ima vrednost NULL ne kaže nikamor.

Oglejmo si primer s simbolom NULL:

```
// Program 4
#include <stdio.h>

int main()
{
   // inicializacija kazalca
   int *k_stevilo = NULL;
   int stevilo = 12;
   int drugo = 5;
   int **k_k_stevilo = &k_stevilo;

   printf("Zacetna vrednost kazalca         : 0x%X\n", k_stevilo);

   // 1. korak: naslov spremenljivke stevilo
   k_stevilo = &stevilo;
   printf("Naslov spremenljivke stevilo(%d) : 0x%X\n", *k_stevilo,
                                              k_stevilo);

   // 2. korak: naslov spremenljivke drugo
   k_stevilo = &drugo;
   printf("Naslov spremenljivke drugo(%d)    : 0x%X\n", *k_stevilo,
                                              k_stevilo);

   // 3. korak: naslov kazalca k_stevilo
   printf("Naslov kazalca k_stevilo         : 0x%X\n", k_k_stevilo);

   // 4. korak: naslov kazalca k_k_stevilo
   printf("Naslov kazalca k_k_stevilo       : 0x%X\n", &k_k_stevilo);

   return 0;
}
```

```
Zacetna vrednost kazalca         : 0x000000
Naslov spremenljivke stevilo(12) : 0x63FDFC
Naslov spremenljivke drugo(5)    : 0x63FDF8
Naslov kazalca k_stevilo         : 0x63FE00
Naslov kazalca k_k_stevilo       : 0x63FDF4
```

Analizirajmo delovanje programa s pomočjo slike 6. Najprej definiramo kazalec `k_stevilo` in ga inicializiramo z `NULL`. Sledi definicija in inicializacija spremenljivke `stevilo` (1. korak). Naslov spremenljivke izpišemo tako, da ga priredimo v kazalec `k_stevilo` in izpišemo kazalec. Isto napravimo še za spremenljivko `drugo` (2. korak).



Slika 6 (spreminjanje vrednosti kazalca)

Dereferenciranje kazalca

Znaka & in * imata dve različni vlogi. Kadar je pred njima zapisan podatkovni tip (na primer int* in float&), z njima definiramo kazalec oziroma referenco. Kadar pa nastopata v izrazih, pa imata vlogo operatorja.

Operator * uporabljamo, če želimo preko kazalca dostopati do vrednosti, na katero kazalec kaže. V tem primeru uporabimo pred kazalcem prefiksni operator *, ki vrne to vrednost. To imenujemo tudi **dereferenciranje** (dereferencing) kazalca. Operand operatorja * mora biti torej kazalec.

Definirajmo spremenljivko stevilo in kazalec nanjo:

```
int stevilo = 12;
int *k_stevilo = &stevilo;
```

Tako spremenljivka `stevilo` kot izraz `*k_stevilo` vsebujeta vrednost 12. Vidimo, da je v tem primeru `*k_stevilo` drugo ime (alias) za spremenljivko `stevilo`.

Operatorja * in & sta komplementarna. Če uporabimo oba hkrati nad istim izrazom, ostane izraz nespremenjen.

Spomnimo se še kazalca `k_k_stevilo`. Če nad njim uporabimo operator *, dobimo kazalec, ki ga lahko ponovno dereferenciramo in dobimo vrednost spremenljivke.

```
// Program 5
#include <stdio.h>

int main()
{
   int stevilo = 12;
   int *k_stevilo = &stevilo;
   int **k_k_stevilo = &k_stevilo;

   printf("Slika pomnilnika:\n");

   printf("               naslov  | vrednost\n");
   printf("             ---------------------\n");
   printf("k_k_stevilo   0x%X | 0x%X\n", &k_k_stevilo, k_k_stevilo);
   printf("k_stevilo     0x%X | 0x%X\n", &k_stevilo,     k_stevilo);
   printf("stevilo       0x%X | %d\n\n", &stevilo,         stevilo);

   printf("Dereferenciranje kazalca:\n\n");
   printf("  k_k_stevilo: 0x%X\n",   k_k_stevilo);
   printf(" *k_k_stevilo: 0x%X\n",  *k_k_stevilo);
   printf("**k_k_stevilo: %d\n",   **k_k_stevilo);

   return 0;
}
```

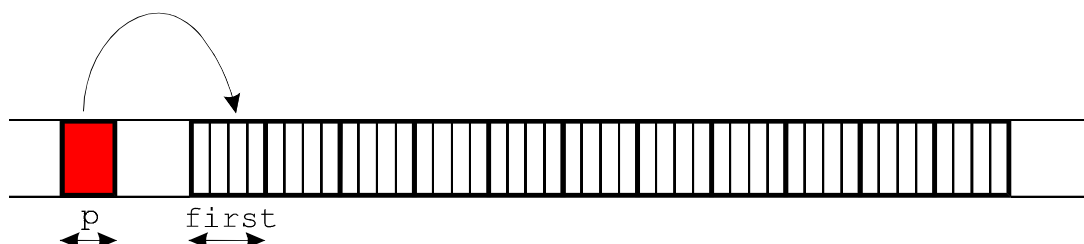
Rezultat:

```
Slika pomnilnika:
               naslov  | vrednost
             ---------------------
k_k_stevilo   0x63FDF8 | 0x63FDFC
k_stevilo     0x63FDFC | 0x63FE00
stevilo       0x63FE00 | 12

Dereferenciranje kazalca:

  k_k_stevilo: 0x63FDFC
 *k_k_stevilo: 0x63FE00
**k_k_stevilo: 12
```

Kot smo že povedali, povečevanje kazalca za 1 torej pomeni, da poslej kaže na naslednji objekt tega tipa. Ta lastnost nam daje tudi idejo, kako urediti delo z vektorji. Vzemimo, da imamo v pomnilniku prostor za 10 realnih števil in da je na začetku tega prostora spremenljivka z imenom `first`. Potem definiramo kazalec:

```
float *p;
p = &first;
```

Dobimo:

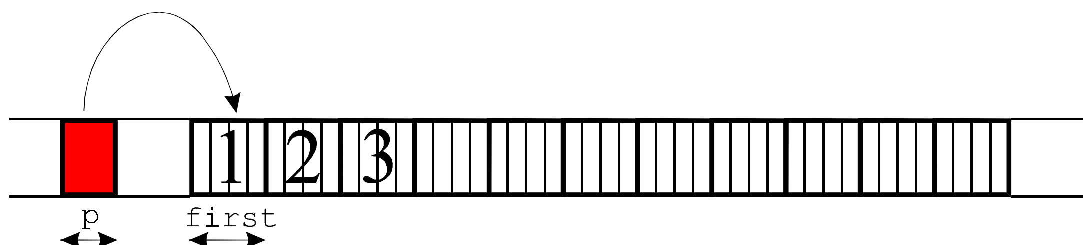


Slika 7 (slika v pomnilniku po zgornjih ukazih)

Z elementi vektorja bi zdaj lahko delali takole:

```
*(p+0) = 1;
*(p+1) = 2;
*(p+2) = 3;
```

in v pomnilniku bi dobili takole stanje:



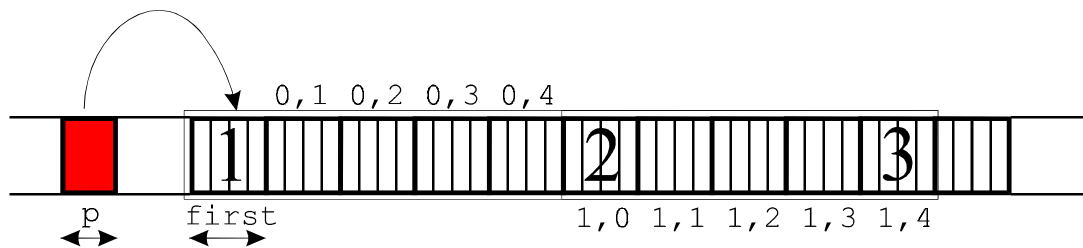
Slika 8 (stanje v pomnilniku po prirejanju)

Namesto kot vektor bi isti pomnilnik lahko razlagali kot dvodimenzionalno polje z dvema vrsticama in petimi stolpci. Če še vedno velja:

```
float *p;
p = &first;
```

Potem bi s posameznimi elementi delali tako (štejemo tako, da imata prva vrstica in prvi stolpec indeksa 0):

```
*(p+0*5+0) = 1;   // element (0,0)
*(p+1*5+0) = 2;   // element (1,0)
*(p+2*5+4) = 3;   // element (1,5)
```



Slika 9 (pomnilnik razumemo kot dvodimenzionalno matriko)

Polja

Gornji način dela je ostal v pogojniku zaradi tehle slabosti:

- ne znamo narediti prostora za vektor,
- način doseganja podatkov je sintaktično neudoben.

Ker se vektorji in matrike v programih pogosto uporabljajo, so avtorji C-ja predvideli operator za delo s polji, ki pomaga narediti prostor in olajša doseganje elementov. C pa ne pozna tipa »polje« niti ne zna delati s poljem kot celoto, kar je v primerjavi z drugimi jeziki primitivno.

Polje z desetimi elementi (z indeksi od 0 do 9) bi naredili in uporabljali tako:

```
float p[10];
p[0] = 1;
p[1] = 2;
p[2] = 3;

p = p+1;    // prepovedano
```

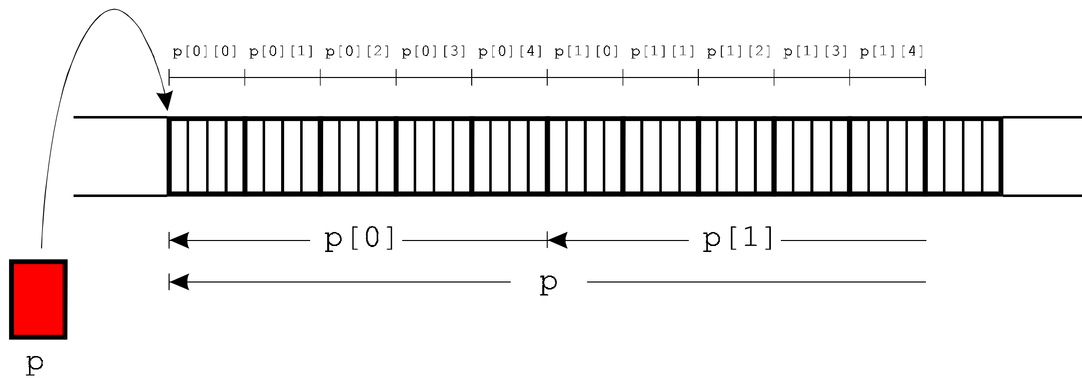
Zadnji ukaz ni dovoljen, ker `p` ni kazalec, ampak polje. Nikjer v pomnilniku ni prostora za naslov začetka polja, zato tja tudi ne moremo prištevati vrednosti.

Operator [] ima dva operanda kazalec (ali polje) in celo število, njegov rezultat pa je identičen temu, kar smo z vektorji počeli v gornjem poglavju.

Večdimenzionalna polja naredimo tako, da naredimo polje manj-dimenzionalnih:

```
float p[2][5];
```

`p` je polje s dvema elementoma, ki so polja s po petimi `float` vrednostmi. V pomnilniku je to zloženo kot kaže slika 10:



Slika 11

Slika 11:
Dvodimenzionalno polje, kot je zloženo v pomnilniku. Kotnice pod pomnilnikom imajo namenoma samo eno puščico. Ta kaže na naslov v pomnilniku, ki se razume, če zapišemo kotirano vrednost. `p[0]` je torej istočasno polje (petih elementov) in element (lahko zapišemo `p[0]+1`, kar je identično `p[1]`). `p` ni kazalec (ni v pomnilniku ampak črtkan nekje zunaj), se pa v vsem obnaša tako, kot da bi bil kazalec, le prirejati mu ni mogoče.

Polja inicializiramo tako, da vrednosti elementov naštejemo znotraj zavitih oklepajev:

```
float p[2][5] = {{1,2,-1,3,4.2},{2,1,3,0,0}};
float p[][] = {{1,2,3},{4,5,6}};
```

V zadnjem primeru prevajalnik sam prešteje in ugotovi dimenzije.

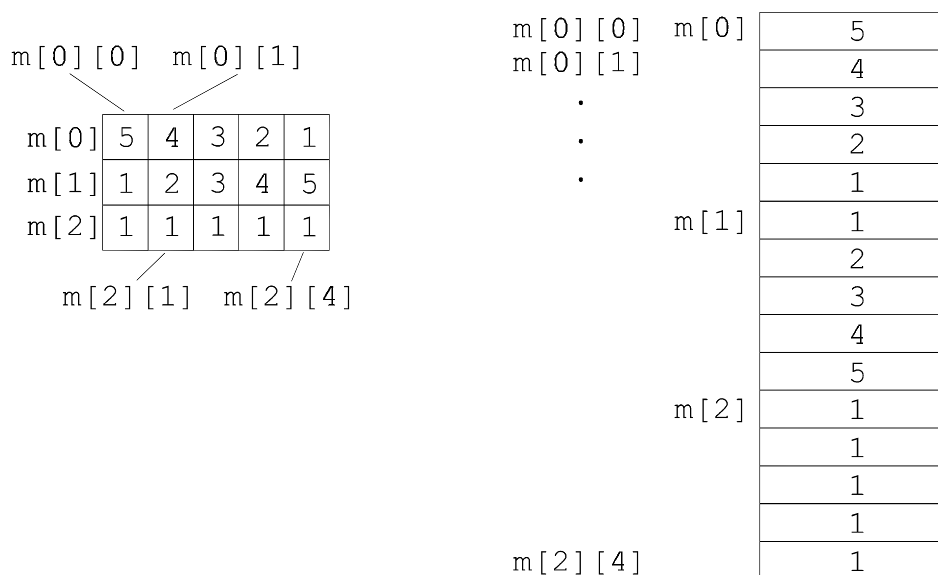
Oglejmo si primer definicije in inicializacije ter izpis vrednosti in vsote elementov matrike:

```
// Program 6
#include <stdio.h>

int main()
{
   int i,j;
   double vsota = 0.0;

   // inicializacija matrike
   double m[3][5] = {{5,4,3,2,1},
                     {1,2,3,4,5},
                     {1,1,1,1,1}};

   // izpis matrike
   for (i = 0; i < 3; i++)
   {
      for (j = 0; j < 5; j++) printf("%1.0f ", m[i][j]);
      printf("\n");
   }

   // prazna vrstica
   printf("\n");
```

```
    // izracun in izpis vsote elementov matrike

    for (i = 0; i < 3; i++)
    {
       for (j = 0; j < 5; j++) vsota += m[i][j];
    }

    printf("Vsota elementov matrike je %2.0f\n", vsota);

    return 0;
}
```

```
5 4 3 2 1
1 2 3 4 5
1 1 1 1 1

Vsota elementov matrike je 35
```

Matrika `m` zaseda v pomnilniku `3 * 5 * sizeof(double)` zlogov. Prostor v pomnilniku je rezerviran zaporedno. Prvih pet elementov v pomnilniku predstavlja prvo vrstico (`polje m[0]`), naslednjih pet elementov so druga vrstica (`polje m[1]`) in zadnjih pet elementov predstavlja zadnjo vrstico matrike (`polje m[2]`).



(a) matrika (b) zaporedje pomnilniških celic

Slika 12 (dvodimenzionalno polje)

Nizi so polja znakov

C nima vgrajenega posebnega tipa za delo z nizi znakov. Pozna samo spremenljivko tipa `char`, nize pa dobimo tako, da naredimo polje znakov, npr. tako:

```
char s[3];
s[0] = 'b';
s[1] = 'l';
s[2] = 'a';
```

Težava pri tej obliki je, da ni nikjer zapisano, kako dolg je niz, ampak se to ve samo tak, kjer je niz deklariran. Če bi imeli npr. funkcijo:

```
void izpisi ( char[?] ) { /* ... */ }
```

bi imeli samo dve možnosti. Ali namesto [?] zapišemo [3] in bo funkcija znala delati samo z nizi, ki so dolgi natanko 3 znake, ali pa da dolžino niza zakodiramo kako drugače. V C-ju velja dogovor, da na koncu niza zapišemo znak '`\0`', torej bi to naredili tako:

```
char s[3];
s[0] = 'b';
s[1] = 'l';
s[2] = 'a';
s[3] = '\0';
```

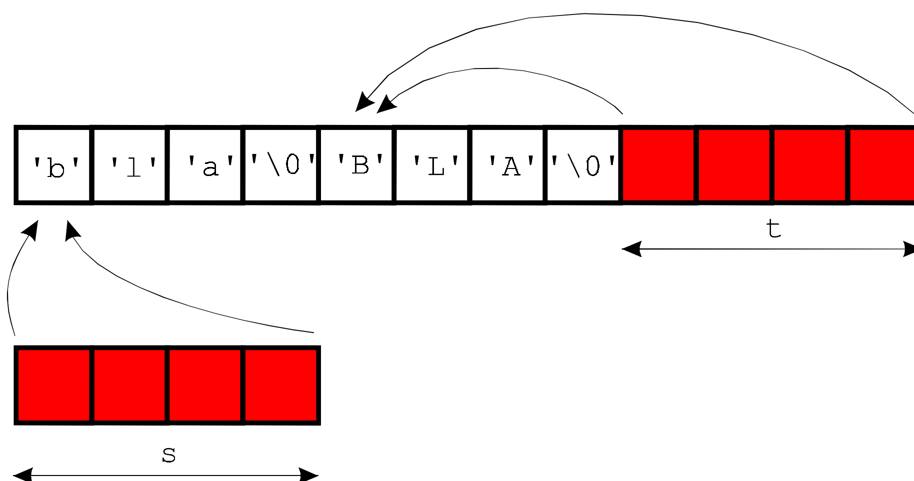
Tak način inicializacije nizov je nepraktičen, zato je na voljo enostavnejši:

```
char s[] = ″bla″;
```

ali

```
char *t = ″BLA″;
```

Pomnilnik po teh definicijah je na sliki 13:



Slika 13 (niz kot polje (s) in niz, dosegljiv preko kazalca (t))

Ker so nizi polja, C nad nizi ne pozna prav nobenih posebnih operacij, na voljo pa je kup funkcij, ki imajo za parameter kazalce na znake.

Strukture

Sestavljenim tipom v C-ju pravimo strukture, kje drugje morda zapisi (record). Strukturo definiramo tako, da ji damo ime in naštejemo, kateri podatkovni elementi (podatkovni člani) jo sestavljajo:

```
struct tocka3d {
      float x;
      float y;
      float z;
};
```

To je bila deklaracija novega tipa, in sicer strukture `tocka3d`, ki ima tri elemente. Spremenljivko tega tipa, z imenom `t1`, naredimo (definiramo) tako:

```
struct tocka3d t1;
```

Ali pa vse hkrati:

```
struct tocka3d {
      float x;
      float y;
      float z;
} t1;
```

Inicializacija je možna, podobno kot pri poljih, z zavitimi oklepaji:

```
struct tocka3d t0 = { 0, 0, 0 };
```

`t0` je na koordinatah 0,0,0. Na posamezne elemente strukture se sklicujemo s pomočjo operatorja '.'. Po gornji definiciji bi točko `t0` premaknili na koordinate (1,2,-1) takole:

```
t0.x = 1;
t0.y = 2;
t0.z = -1;
```

Ime novega tipa lahko stoji povsod tam, kjer lahko stoji ime vgrajenega tipa. 'float' ali pa 'struct tocka3d' sta sintaktično enaka. Tudi kazalce na strukture imamo lahko:

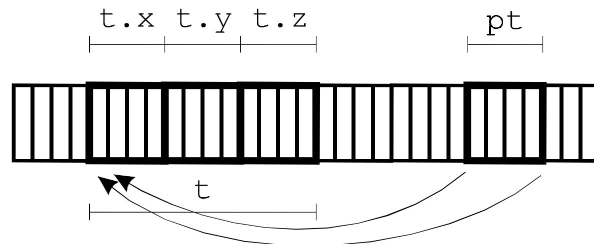
```
struct tocka3d t;
struct tocka *pt;
pt = &t;
```

in potem:

```
(*pt).x = 1;
```

Zgornja sintaksa je okorna, zato ima C za doseganje elementov strukture prek kazalcev bližnjico v obliki novega operatorja `->`. Zadnjo vrstico bi napisali tako:

```
pt -> x = 1;
```



Slika 14 (slika pomnilnika s strukturo t in kazalcem pt, ki kaže na t)

LITERATURA:

- Žiga Turk: UVOD V OBJEKTNO ORIENTIRANO PROGRAMIRANJE IN JEZIK C++; DZS Ljubljana, 1991
- V. Žumer, N. Korbar: Programiranje v jeziku C++; FERI Maribor, 1997
- Brian W. Kernighan, Dennis M. Ritchie: Programski jezik C; FER Ljubljana, 1991